Методичка 8.3 - Знакомство с дизассемблером IDA. Часть 2

Обзор базовых функций IDA Freeware

Возвращаемся к функции main программы prog-6.1.exe.

В самом начале функции мы видим, что IDA добавила 4 строки перед началом инструкций:

```
var_38= byte ptr -38h
var_30= byte ptr -30h
arg_0= dword ptr  8
arg_8= qword ptr  10h
```

Это переменные, которые обнаружила IDA в процессе автоматического анализа. Если название начинается на var, значит это локальная переменная в функции, а если название начинается на arg, то это входные аргументы функции. Если исходя из кода мы можем понять, что хранится в какой-то переменной, то ее тоже нужно переименовать.

Переименование функций и переменных очень полезно использовать при анализе бинарных файлов. Чем больше функций мы сможете определить, тем легче будет понять как работает программа.

Давайте попробуем посмотреть на код функции main и понять, что там происходит:

В самом начале мы видим подготовку стека:

```
mov     [rsp+arg_8], rdx
mov     [rsp+arg_0], ecx
push    rsi
push    rdi
```

Затем происходит выделение памяти в стеке.

```
mov    eax, 48h
call ...
sub     rsp, 48h
```

Далее записываем адрес какой-то переменной в регистр RAX, пока не понятно, что это за переменная.

```
lea     rax, [rsp+58h+var_30]
```

Затем видим запись адреса строки “AABBCCDDEEFF” в регистр RCX.

```
lea     rcx, aAabbccddeeff
```

Здесь мы видим копирование адресов из RAX и RCX в RDI и RSI.

```
mov     rdi, rax
mov     rsi, rcx
```

А здесь мы задаем счетчик (0x0D (13) (строка заканчивается нулевым байтом) - длина строки) для следующей инструкции (rep movsb), которая записывает байты из одной строки в другую.

```
mov     ecx, 0Dh
rep     movsb
```

Код, который мы сейчас проанализировали отвечает за назначение какой-то переменной строки “AABBCCDDEEFF”, но пока мы не знаем, что это за значение, поэтому не можем переименовать переменную var_30.

Далее мы видим проверки на количество аргументов. Аргумент arg_0, это argc. Можем переименовать arg_0 в argc.

Значение переменной argc сравнивается с 2.

```
cmp     [rsp+58h+argc], 2
jz      short loc_140001060
```

Если сравнение не проходит, то мы переходим коду, который печатает подсказку и переходит к завершению программы.

Если сравнение проходит, то мы переходим к другому блоку кода.

Видим, что в коде используется аргумент arg_8, давайте переименуем его в argv.

Далее видим вызов неизвестной нам функции. Вспоминаем, что в x64 первый аргумент передается через регистр RCX. Функция отрабатывает, значение, которое вернула функция должно быть в регистре RAX.

С помощью инструкции MOV значение сохраняется в стеке.

Далее в регистр RCX кладется указатель на строку “AABBCCDDEEFF”, это переменная var_30.

Вызывается такая же функция. Результат кладется в регистр RAX. Затем из стека забирается значение, которое было ранее сохранено и оно кладется в регистр RCX. И затем происходит сравнение на равенство с помощью условного перехода JZ.

Если сравнение проходит неудачно, то нас перекидывает на код, который сообщает о том, что длина ключа должна быть 12 символов.

После этой строки мы видим вызов какой-то функции, это скорее всего printf. Переименуем ее. Затем происходит прыжок к завершению функции.

Исходя из строки ”License key should contains 12 symbols.” можно сделать вывод, что ранее проверялась длина строки. Давайте переименуем функцию - strlen.

Если длина ключа оказалась равно 12, то мы переходи на другой блок кода.

С помощью инструкций MOV и IMUL считается номер элемента массива argv.

mov eax, 8

Это размер адреса. Напомню, что argv это массив указателей. В программах с архитектурой x64 размер указателя составляет 8 байт, так как адрес имеет размер 64 бита.

Далее с помощью инструкции IMUL считается номер элемента в массиве.

imul rax, 1

В данном случае мы получаем указатель на первый элемент массива.

Затем мы загружаем в регистр RDX указатель на строку “AABBCCDDEEFF”, а далее загружаем в регистр RCX указатель на первый элемент массива argv.

Затем происходит вызов какой-то функции. Далее происходит проверка на ноль. И в зависимости от результат, ключ либо правильный, либо нет.

Очень похоже на то, что перед сравнением вызывается функция strcmp, так как именно она возвращает нулевое значение, если строки равны. Переименуем функцию.

Исходя из той информации, которую мы получили, можно сделать вывод, что программа принимает единственный входной аргумент и сравнивает его со строкой “AABBCCDDEEFF”. Давайте это проверим.

> prog-6.1.exe AABBCCDDEEFF

Congratulations! License key is correct.

Ключ подходит.

Таким образом, в процессе анализа программы, выяснили корректный ключ. Программа была достаточно простой, поэтому мы не испытали никаких трудностей в процессе анализа, но так бывает не всегда.

Если у вас не получается понять, как программа проверяет ключ, но вы нашли функцию триггер (функция, которая принимает решение о корректности ключа), то пропатчить программу так, чтобы она всегда выдавала то, что нам нужно.

Давайте попробуем пропатчить программу. Возвращаемся к участку кода, где происходит сравнение значения, которое вернула функция strcmp.

```
call    strcmp
test    eax, eax
jz      short loc_1400010B8
```

Помним, если 0, то прыжок, если не 0, то прыжка не будет. Наша цель сделать так, чтобы прыжок был при неправильном ключе. Один из вариантов, это заменить инструкцию JZ на JNZ, т.е. прыгать тогда, когда значение не 0.

Нам нужно найти в коде опкод инструкции JZ, и заменить на на опкод инструкции JNZ.

Выбираем нашу инструкцию JZ и на свободном месте с помощью правой кнопкой мыши вызываем меню и выбираем Text view. Для того, чтобы вернуться в режим графа - Graph View.

```
.text:0000000140001090          call    strcmp
.text:0000000140001095          test    eax, eax
.text:0000000140001097          jz      short loc_1400010B8
```

В этом режиме мы можем увидеть опкоды, но для этого их нужно включить в настройках IDA.

Options - General - Number of opcode bytes - 8

```
.text:0000000140001090 E8 AB 88 00 00          call    strcmp
.text:0000000140001095 85 C0                   test    eax, eax
.text:0000000140001097 74 1F                   jz      short loc_1400010B8
```

Теперь мы видим, что инструкция “jz   short loc_1400010B8” имеет опкод - 741F. Что это значит?

Первый байт - это опкод команды JZ, второй байт - это смещение относительно следующей инструкции, куда нужно прыгать.

Смещение нам менять не нужно, а нужно заменить только опкод JZ (который 74) на JNZ (?). Чтобы его узнать можно обратиться в гугл с запросом - “jnz opcode x86”. Быстро находим, что опкод JNZ - это 75.

Теперь нам нужно заменить байт 0x74 на байт 0x75. Выделяем нашу инструкцию JZ.

Edit - Patch Program - Change Byte

Меняем 0x74 на 0x75 и нажимаем OK. Видим, что теперь инструкция отображается как JNZ.

```
.text:0000000140001097 75 1F                    jnz     short loc_1400010B8
```

Edit - Patch Program - Apply Patches to input file…

Рекомендую поставить галочку - Create backup и нажимаем OK. Проверяем.

> prog-6.1.exe AAAAAAAAAAAA

Congratulations! License key is correct.

Отлично, теперь программа принимает некорректные ключи. Патчинг прошел удачно.

После завершения работа с файлом IDA предложит сохранить вашу работу в виде проекта, чтобы не пришлось повторять проделанную работу.

Нужно выбрать Pack Database (Store).

Для того, чтобы открыть проект мы можем дважды кликнуть по нему.

Дизассемблер IDA имеет встроенный отладчик. Давайте посмотрим, как она работает.

Переходим в режим Text View. Ставим точку останова, например, в начале функции main и запускаем отладчик.

Видим, как меняются значения в регистрах.

Функция перешла к завершению, так как у нее нет входных параметров. Давайте добавим входной аргумент. (Debugger - Process Options - Parameters).

Проходим снова.